Abstractions for Fault–Tolerant Distributed System Verification*

Lee Pike[1], Jeffrey Maddalon[1], Paul Miner[1], and Alfons Geser[2]

[1] Formal Methods Group
NASA Langley Research Center
M/S 130, Hampton, VA 23681-2199
{lee.s.pike, j.m.maddalon, paul.s.miner}@nasa.gov
[2] National Institute of Aerospace
144 Research Drive, Hampton, VA 23666
geser@nianet.org

Summary. Four kinds of abstraction for the design and analysis of fault–tolerant distributed systems are discussed. These abstractions concern system messages, faults, fault–masking voting, and communication. The abstractions are formalized in higher–order logic, and are intended to facilitate specifying and verifying such systems in higher–order theorem–provers.

1 Introduction

In recent years, we have seen tremendous growth in the development of embedded computer systems with critical safety requirements [10, 12], and there is no expectation that this trend will abate. For instance, steer–by–wire systems are currently being pursued [11]. To withstand faulty behavior, safety–critical systems have traditionally employed analog backup systems in case the digital system fails; however, many new "by–wire" systems have no analog backup. Instead, they rely on integrated digital fault–tolerance.

Due to their complexity and safety–critical uses, fault–tolerant embedded systems require the greatest assurance of design correctness. One means by which a design can be shown correct is formal methods. Formal methods are especially warranted if we recall that published and peer–reviewed informal proofs–of–correctness of seemingly simple fault–tolerance algorithms have been incorrect [16]. Here, we focus on formal methods involving higher–order theorem–provers.

Although many fault–tolerant distributed systems and algorithms have been specified and verified, the abstractions used have often been *ad–hoc* and system–specific. Developing appropriate abstractions is often the most difficult and time–consuming part of formal methods [25]. We present these abstractions to systematize and facilitate the practice of abstraction.

*Accepted at *Theorem-Proving in Higher-Order Logics* (TPHOLs), 2004.

The abstractions presented are in the spirit of abstractions of digital hardware developed by Thomas Melham [19, 18]. They are intended to make specifications and their proofs of correctness less tedious [14], less error–prone, and more consistent. Although the abstractions we describe are quite general, we intend for them to be accessible to the working verification engineer.

These abstractions are the outcome of the on–going project "Scalable Processor–Independent Design for Electromagnetic Resilience" (SPIDER) at NASA's Langley Research Center and the National Institute of Aerospace. One of the project goals is to specify and verify the Reliable Optical Bus (ROBUS), a state–of–the–art fault–tolerant communications bus [27, 20]. SPIDER is the basis of an FAA study exploring the use of formal methods, especially theorem–proving, in avionics certification. The abstractions have proved useful in this project, and in fact are the basis of a generalized fault–tolerant library of PVS theories mentioned in Sect. 8.

The structure of our paper is as follows. We discuss fault–tolerant distributed systems in Sect. 2. Section 3 gives an overview of the four abstractions presented in this paper. Sections 4 through 7 explain these abstractions. Each section presents an abstraction, and then the abstraction is formalized in higher–order logic. We provide some concluding remarks and point toward future work in the final section.

2 Fault–Tolerant Distributed Systems

Introductory material on the foundations of distributed systems and algorithms can be found in Lynch [17]. Some examples of systems that have fault–tolerant distributed implementations are databases, operating systems, communication busses, file systems, and server groups [3, 27, 2].

A *distributed system* is modeled as a graph with directed edges. Vertices are called *processes*. Directed edges are called *communication channels* (or simply *channels*). If channel c points from process p to process p', then p can send messages over c to p', and p' can receive messages over c from p. In this context, p is the *sending process* (or *sender*) and p' is the *receiving process* (or *receiver*). Channels may point from a process to itself. In addition to sending and receiving messages, processes may perform local computation.

A *fault–tolerant* system is one that continues to provide the required functionality in the presense of faults. One way to implement a fault–tolerant system is to use a distributed collection of processes such that a fault that affects one process will not adversely affect the whole system's functionality. This type of system is referred to as a *fault–tolerant distributed system*.

3 Four Kinds of Abstraction

We introduce four fundamental abstractions in the domain of fault–tolerant distributed systems. *Message Abstractions* address the kinds of messages sent and received. We consider abstractions both of the content of messages and of the faultiness of messages. *Fault Abstractions* address the kinds of faults possible as well as their affects in the system. *Fault–Masking Abstractions* address the kinds of local computations processes make to mask faults. Finally, *Communication Abstractions* address

the kinds of messages communicated and the properties required for communication to succeed in the presence of faults.

Our formal expressions are stated in the language of higher–order functions: variables can range over functions, and functions can take other functions as arguments. Furthermore, we use uninterpreted functions (i.e., functions with no defining body) that act as constants when applied to their arguments. Curried functions and lambda abstraction are also used. For a brief overview of higher–order logic from a practitioner's perspective, see, for example, Melham [19] or the PVS language reference [9]. A small datatype, fully explained in Sect. 4, is also used.

The abstractions have all been formalized in the Prototype Verification System (PVS), a popular interactive industrial–strength theorem proving system [21, 8]. They are available at [23].

4 Abstracting Messages

4.1 Abstraction

Messages communicated in a distributed system are abstracted according to their correctness. We distinguish between *benign messages* and *valid messages*. The former are messages that a non–faulty receiving process recognizes as incorrect; the latter are messages that a non–faulty receiving process does not recognize as incorrect. Note that a valid message may be incorrect: the receiving process just does not *detect* that the message is incorrect.

Benign messages abstract various sorts of misbehavior. A message that is sufficiently garbled during transmission may be caught by an error–checking code [7] and deemed benign. Benign messages also abstract the absence of a message: a receiver expecting a message but detecting the absence of one takes this to be the 'reception' of a benign message. In a synchronized systems with global communication schedules, they abstract messages sent and received at unscheduled times.

4.2 Formalization

Let the set MSG be a set of messages of a given type. We define a datatype over elements of MSG. The set of all possible datatype elements is $ABSTRACT\_MSG[MSG]$.

The datatype has two constructors, *valid_msg* and *benign_msg*. The former takes an element $m \in MSG$ and creates the datatype element $valid\_msg[m]$. The constructor also has an associated extractor *value* such that

$$value(valid\_msg[m]) = m \ .$$

The other constructor, *benign_msg*, is a constant datatype element; it is a constructor with no arguments. All benign messages are abstracted as a single message; thus, the abstracted incorrect message cannot be recovered. Finally, we define two recognizers, *valid_msg*? and *benign_msg*? with the following definitions. Let $a \in ABSTRACT\_MSG[MSG]$.

$$valid\_msg?(a) \stackrel{\text{df}}{=} \exists m.\, m \in MSG \wedge a = valid\_msg[m] \ ,$$

and

$$benign\_msg?(a) \stackrel{\text{df}}{=} a = benign\_msg\ .$$

We summarize this datatype in Fig. 1. Let $m \in MSG$.

Constructors	Extractors	Recognizers
$valid\_msg[m]$	$value$	$valid\_msg?$
$benign\_msg$	none	$benign\_msg?$

Fig. 1. Abstract Messages Datatype

5 Abstracting Faults

There are two closely related abstractions with respect to faults. The first abstraction, *error types*, partitions the possible locations of faults. The second abstraction, *fault types*, partitions faults according to the manifestation of the errors caused by the faults.[3]

5.1 Abstracting Error Types

Picking the right level of abstraction and the right components to which faults should be attributed is a modeling issue that has been handled in many different ways. We think this is a particularly good example of the extent to which modeling choices can affect specification and proof efficacy.

Both processes and channels can suffer faults, and there are fault-tolerant algorithms tolerating faults of each kind [17]. In actual systems, both may be faulty, but reasoning about process and channel faults together is tedious. Fortunately, such reasoning is redundant – channel faults can be abstracted as process faults. A channel between a sending process and a receiving process can be abstracted as being an extension either of the sender or of the receiver. For instance, a lossy channel abstracted as an extension of the sender is modeled as a process failing to send messages.

Even if we abstract all faults to ones affecting processes and not channels, we are left with the task of abstracting how the functionality of a process – sending, receiving, or computing – is degraded. One possibility is to consider a process as an indivisible unit so that a fault affecting one of its functions is abstracted as affecting its other functions, too. Another possibility is to suppose all processes perform local computation correctly regardless of their faultiness, such as in the models used in [26, 22]. Finally, models implicit in [5, 16] abstract process faults as being ones affecting *only* a process' ability to send messages. So even if a fault affects a process' ability to receive messages or compute, the fault is abstractly propagated to a fault affecting the process' ability to send messages.

[3] An *error* is "that part of the system state which is *liable to lead to subsequent failure*," while a *fault* is "the *adjudged or hypothesized cause* of an error" [15].

All three models above are *conservative*, i.e., the abstraction of a fault is at least as severe as the fault. This is certainly true of the first model in which the whole process is considered to be degraded by any fault, and it is true for the second model, too. Even though it is assumed that a process can always compute correctly, its computed values are inconsequential if it can neither receive nor send correct values. As for the third model, the same reasoning applies – even if a faulty process can receive messages *and* compute correctly, it cannot send correct messages to other processes.

The model we choose is one in which all faults are abstracted to be ones degrading send functionality, and in which channels are abstracted as belonging to the sending process. There are two principal advantages to this model. First, the model allows us to disregard faults when reasoning about the ability of processes to receive and compute messages. Second, whether a message is successfully communicated is determined solely by a process' send functionality.

5.2 Abstracting Fault Types

Faults result from innumerable occurrences including, physical damage, electromagnetic interference, and "slightly–out–of–spec" communication [4]. We collect these fault occurrences into *fault types* according to their effects in the system.

We adopt the *hybrid fault model* of Thambidurai and Park [28]. A process is called *benign*, or *manifest*, if it sends only benign messages, as described in Sect. 4. A process is called *symmetric* if it sends every receiver the same message, but these messages may be incorrect. A process is called *asymmetric*, or *Byzantine* [13], if it sends different messages to different receivers. All non-faulty processes are also said to be *good*.

Other fault models exist that provide more or less detail than the hybrid fault model above. The least detailed fault model is to assume the worst case scenario, that all faults are asymmetric. The fault model developed by Azadmanesh and Kieckhafer [1] is an example of a more refined model. All such fault models are consistent with the other abstractions in this paper.

5.3 Formalization

We begin by formalizing fault types. Let S be the set of sending processes. Let *asym*, *sym*, *benign*, and *good* be constants representing the fault types asymmetric, symmetric, benign, and good, respectively.

As mentioned, we abstract all faults to ones that affect a process' ability to send messages. To model this formally, we construct a function modeling a process sending a message to a receiver. The range of the function is the set of abstract messages, elements of the datatype defined in Sect. 4. As explained, MSG is a set of messages, and $ABSTRACT\_MSG[MSG]$ is the set of datatype elements parameterized by MSG. Let $s \in S$ and $r \in R$ be a sending and receiving process respectively. Let $msg\_map : S \rightarrow MSG$ be a function from senders to the message they intend to send, and let $sender\_status$ be a function mapping senders to their fault partition. The function outputs the abstract message received by r from s.

$$send(msg\_map, sender\_status, s, r) \stackrel{\text{df}}{=} \begin{cases} valid\_msg[msg\_map(s)] & : \; sender\_status(s) = good \\ benign\_msg & : \; sender\_status(s) = ben \\ sym\_msg(msg\_map(s), s) & : \; sender\_status(s) = sym \\ asym\_msg(msg\_map(s), s, r) & : \; sender\_status(s) = asym \; . \end{cases}$$

If s is good, then r receives a valid abstract message from s. If s is benign, then r receives a benign message. In the last two cases – in which s suffers a symmetric or asymmetric fault – uninterpreted functions are returned. Applied to their arguments, sym_msg and $asym\_msg$ are unknown abstract message constants. The function $asym\_msg$ models a process suffering an asymmetric fault by taking the receiver as an argument: for receivers r and r', $asym\_msg(msg\_map(s), s, r)$ is not necessarily equal to $asym\_msg(msg\_map(s), s, r')$. On the other hand, the function sym_msg does not take a receiver as an argument, so all receivers receive the same abstract message from a particular sender.

6 Abstracting Fault–Masking

6.1 Abstraction

Some of the information a process receives in a distributed system may be incorrect due to the existence of faults as described in Sect. 5. A process must have a means to mask incorrect information generated by faulty processes. Two of the most well–known are (variants of) a majority vote or a middle–value selection, as defined in the following paragraph. These functions are similar enough to abstract them as a single fault–masking function.

A majority vote returns the majority value of some multiset (i.e., a set in which repetition of values is allowed), and a default value if no majority exists. A middle–value selection takes the middle value of a linearly–ordered multiset, if the cardinality of the multiset is odd. If the cardinality is an even integer n, then the natural choices are to compute one of (1) the value at index $\lfloor n/2 \rfloor$, (2) the value at index $\lceil n/2 \rceil$, or (3) the average of the two values from (1) and (2). Of course, these options may yield different values; in fact, (3) may yield a value not present in the multiset.

For example, for the multiset $\{1, 1, 2, 2, 2, 2\}$, the majority value is 2, and the middle–value selection is also 2 for any of the three ways to compute the middle–value selection. Both the majority vote and the middle–value selection yield the same value for the above multiset. For any multiset that can be linearly–ordered, if a majority value exists, then the majority value is equal to the middle–value selection (for any of the three ways to compute it mentioned above).

The benefit of this abstraction is that we can define a single fault–masking function (we call it a *fault–masking vote*) that can be implemented as either a majority vote or a middle–value selection (provided the data over which the function is applied is linearly–ordered).

This allows us to model what are usually considered to be quite distinct fault–tolerant distributed algorithms uniformly. Concretely, this abstraction, coupled with the other abstractions described in this paper, allow certain clock synchronization algorithms (which usually depend on a middle–value selection) and algorithms in the spirit of an Oral Messages algorithm [13, 16] (which usually depend on a majority vote) to share the same underlying models.

6.2 Formalization

The formalization we describe models a majority vote and a middle–value selection over a multiset. A small lemma stating their equivalence follows. Definitions of standard and minor functions are omitted.

Based on the NASA Langley Research Center PVS *Bags* Library [6], a multiset is formalized as a function from values to the natural numbers that determines how many times a value appears in the multiset (values not present are mapped to 0). Thus, let V be a nonempty finite set of values[4], and let $ms : V \to \mathbb{N}$ be a multiset.

To define a majority vote, we define the cardinality of a multiset ms to be the summation of value–instances in it:

$$|ms| \stackrel{\text{df}}{=} \sum_{v \in V} ms(v) \ .$$

The function maj_set takes a multiset ms and returns the set of majority values in it.

$$maj\_set(ms) \stackrel{\text{df}}{=} \{v \mid 2 \times ms(v) > |ms|\} \ .$$

This set is empty if no majority value exists, or it is a singleton set. Thus, we define $majority$ to be a function returning the special constant $no\_majority$ if no majority value exists and the single majority value otherwise.

$$majority(ms) \stackrel{\text{df}}{=} \begin{cases} no\_majority & : \quad majority?(ms) = \emptyset \\ \epsilon(maj\_set(ms)) & : \quad \text{otherwise} \ . \end{cases}$$

The function ϵ is the choice operator that takes a set and returns an arbitrary value in the set if the set is nonempty. Otherwise, an arbitrary value of the same type is returned [19].

Now we formalize a middle–value selection. Let V have the linear order $\preceq$ defined on it. The function mid_val_set takes a multiset and returns the set of values at index $\lceil n/2 \rceil$ when the values are ordered from least to greatest (we arbitrarily choose this implementation). The set is always a singleton set.

$$mid\_val\_set(ms) \stackrel{\text{df}}{=} \left\{ v \;\middle|\; \begin{array}{l} 2 \times |lower\_filter(ms, v)| > |ms| \wedge \\ 2 \times |upper\_filter(ms, v)| \geq |ms| \end{array} \right\} \ .$$

The function $lower\_filter$ filters out all of the values of ms that are less than or equal to v and $upper\_filter$ filters out the values greater than or equal to v. The function $lower\_filter$ is defined as follows:

$$lower\_filter(ms, v) \stackrel{\text{df}}{=} \lambda i. \begin{cases} ms(i) & : \quad i \preceq v \\ 0 & : \quad \text{otherwise} \ . \end{cases}$$

Similarly,

$$upper\_filter(ms, v) \stackrel{\text{df}}{=} \lambda i. \begin{cases} ms(i) & : \quad v \preceq i \\ 0 & : \quad \text{otherwise} \ . \end{cases}$$

[4] If V is finite, then multisets are finite. Fault–masking votes can only be taken over finite multisets.

The relation $middle\_value?(ms)$ is guaranteed to be a singleton set, so using the function ϵ mentioned above, we can define $middle\_value$ to return the middle value of a multiset:

$$middle\_value(ms) \stackrel{\text{df}}{=} \epsilon(mid\_val\_set(ms)) \ .$$

The following theorem results.

Theorem 1 (Middle Value is Majority) *$majority(ms) \neq no\_majority$ implies $middle\_value(ms) = majority(ms)$.*

7 Abstracting Communication

We identify two abstractions with respect to communication. First, we abstract the kinds of data communicated. Second, we identify the fundamental conditions that must hold for communication to succeed.

7.1 Abstracting Kinds of Communication

Some kind of information can be modelled by a real valued, uniformly continuous function of time. Intuitively, a function is uniformly continuous if small changes in its argument produce small changes in its result; see e.g., Rosenlicht [24]. For example, the values of analog clocks and of thermometers vary with time, and the rate of change is bounded. In a distributed system, a process may *sample* such a function, i.e., determine a digital approximation of the function's value at a given moment. Because communication is not instantaneous, the sample requested and the one received may differ. We therefore call such functions *inexact functions* and the communication of their values *inexact communication*. Other functions, such as an array sorting algorithm, do not depend on time. We call these *exact functions* and communication involving them *exact communication*.

7.2 Abstracting Communication Conditions

Communication in a distributed system is successful if *validity* and *agreement* are guaranteed to hold. For exact communication, their general forms are:

Exact Validity: A good receiver's fault–masking vote is equal to the message good processes send.
Exact Agreement: All good processes have equal fault–masking votes.

For inexact communication we have similar conditions:

Inexact Validity: A good receiver's fault–masking vote is bounded above and below by the messages good processes send, up to a small error margin.
Inexact Agreement: All good processes differ in their fault–masking votes by at most a small margin of error.

A validity property can thus be understood as an agreement between a sender and a receiver, whereas an agreement property is an agreement between the receivers. For lack of space, we limit our presentation to guaranteeing validity. Agreement is treated similarly, and complete PVS formalizations and proofs for both are located at [23].

We distinguish between a *functional model* and a *relational model* of communication. In the former, communication is modeled computationally (e.g., using functions like *send* from Sect. 5). In the latter, conditions on communication are stated such that if they hold, communication succeeds. This section presents these conditions.

We specifically present conditions that guarantee validity holds after a single *broadcast communication round* in which each process in a set sends messages to each process in a set of receivers (a degenerate case is when these are singleton sets modeling point–to–point communication between a single sender and receiver). A functional model of a specific communication protocol can be shown to satisfy these conditions through step–wise refinement.

First we show that a single–round exact communication satisfies validity, provided that three conditions hold: *Majority Good*, *Exact Message Error*, and *Message Agreement*. The three conditions state, respectively, that the majority of the values over which a vote is taken come from good senders; that the message received is the message sent; and that every sender sends the same message. We define the *eligible senders* of a receiver to be the set of senders it believes to be good. This is the set of senders that are admitted to the vote by a receiver; the others are ignored.

For single–round inexact communication, we have validity if two conditions hold: *Majority Good* and *Inexact Message Error*. The *Inexact Message Error* condition specifies the variance allowed between a sampled function before and after communication. For inexact communication, we must allow for a small variance to occur. Let f be a time–dependent function. Suppose a process sends a request for f to be sampled at time t. However, the function f might be sampled a little sooner or later than t given the variable delay in communication; let n be the time at which f is actually sampled. The condition states that n$f(n)$ is no less than $f(t) - \varepsilon_{\mathrm{l}}$ and no greater than $f(t) + \varepsilon_{\mathrm{u}}$, where ε_{l} and ε_{u} are constants. This is represented graphically in Fig. 2.

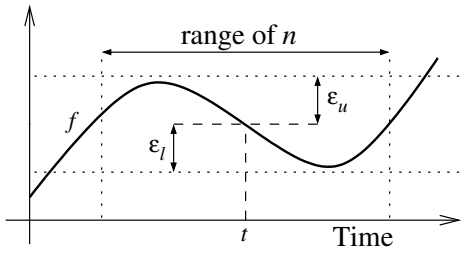

Fig. 2. The Message Error Condition for Inexact Communication

Clock synchronization is an important case of inexact communication. A clock is formalized as a function from real time to clock time. Clocks distributed in the

system need to be synchronized in order to avoid drifting too far apart. As the first step in the synchronization protocol, processes sample one another's clocks. The Inexact Message Error condition abstracts the nominal time each process samples the others' clocks (the time t in the example above) and the variation caused by clocks that have already drifted slightly from each other (the ε_{l} and ε_{u} constants in the example above).

7.3 Formalization for Exact Communication

First we present the model of a round of exact communication. For a single round of communication, let S be the set of senders sending in that round. Let $good\_senders \subseteq S$ be a subset of senders that are good. This set can change as processes become faulty and are repaired, so we treat it as a parameter rather than a constant. For an arbitrary receiver,[5] let $eligible\_senders \subseteq S$ be the set of senders trusted by the receiver. Then the condition Majority Good is defined

$$\begin{aligned} &majority\_good(good\_senders, eligible\_senders) \stackrel{\mathrm{df}}{=} \\ &\quad 2 \times |good\_senders| > |eligible\_senders| \wedge \\ &\quad good\_senders \subseteq eligible\_senders \ . \end{aligned}$$

This stipulates that a majority of the elements in $eligible\_senders$ are elements in $good\_senders$.

Next we describe the values sent and received. Let MSG be the range of the function sampled – these are the values to communicate. The function $ideal : S \rightarrow MSG$ is some function mapping senders to the message they send, thereby freeing us from representing the particular function sampled. Similarly, $actual : S \rightarrow MSG$ maps senders to the message a fixed receiver actually gets from that sender. For exact communication, we require that what is sent is what is received:

$$\begin{aligned} &exact\_message\_error(good\_senders, ideal, actual) \stackrel{\mathrm{df}}{=} \\ &\quad \forall s.\, s \in good\_senders \implies ideal(s) = actual(s) \ . \end{aligned}$$

Message Agreement states that the message content sent by any two senders is the same.

$$\begin{aligned} &message\_agreement(good\_senders, ideal) \stackrel{\mathrm{df}}{=} \\ &\quad \forall s_1, s_2.\, s_1 \in good\_senders \wedge s_2 \in good\_senders \implies ideal(s_1) = ideal(s_2) \ . \end{aligned}$$

Before stating the validity result, we must take care of a technical detail with respect to forming the multiset of messages over which a receiver takes a fault–masking vote. For an arbitrary receiver, let the function $make\_bag$ take as arguments a nonempty $eligible\_senders$ and a function mapping senders to the message the receiver gets. It returns a multiset of received values from senders in $eligible\_senders$.

$$\begin{aligned} &make\_bag(eligible\_senders, actual) \stackrel{\mathrm{df}}{=} \\ &\quad \lambda v. \left| \{ s \mid s \in eligible\_senders \wedge actual(s) = v \} \right| \ . \end{aligned}$$

[5] The receiver can be *any* receiver, good or faulty. The abstractions described in Sect. 5 allow us to ignore the fault status of receivers in formal analysis.

For exact messages, validity is the proposition that the message any good sender sends is the same as the message determined by a fault–masking vote. This proposition is defined as:

$$\begin{aligned} &exact\_validity(eligible\_senders, good\_senders, ideal, actual) \stackrel{\text{df}}{=} \\ &\quad \forall s.\, s \in good\_senders \implies \\ &\qquad ideal(s) = majority(make\_bag(eligible\_senders, actual)) \ . \end{aligned}$$

We use $majority$ for the fault–masking vote, but middle–value selection is acceptable given Thm. 1. Using this proposition the Exact Validity Theorem reads:

Theorem 2 (Exact Validity)
$majority\_good(good\_senders, eligible\_senders) \wedge$
$exact\_message\_error(good\_senders, ideal, actual)) \wedge$
$message\_agreement(good\_senders, ideal)$
implies that
$exact\_validity(eligible\_senders, good\_senders, ideal, actual)$.

7.4 Formalization for Inexact Communication

Next we present the model of a round of inexact communication. The Majority Good condition here is the same as the one for exact communication. We now assume that the elements of MSG have at least the structure of an additive group linearly ordered by $\preceq$. Inexact Message Error is defined as the conjunction of two conditions, *Lower Message Error* and *Upper Message Error*. These two conditions specify, respectively, the maximal negative and positive error that may be introduced by sampling from a good sending process.

$$\begin{aligned} &lower\_message\_error(good\_senders, ideal, actual) \stackrel{\text{df}}{=} \\ &\quad \forall s.\, s \in good\_senders \implies ideal(s) - \varepsilon_{\text{l}} \preceq actual(s) \ ; \end{aligned}$$

$$\begin{aligned} &upper\_message\_error(good\_senders, ideal, actual) \stackrel{\text{df}}{=} \\ &\quad \forall s.\, s \in good\_senders \implies actual(s) \preceq ideal(s) + \varepsilon_{\text{u}} \ ; \end{aligned}$$

$$\begin{aligned} &inexact\_message\_error(good\_senders, ideal, actual) \stackrel{\text{df}}{=} \\ &\quad lower\_message\_error(good\_senders, ideal, actual) \wedge \\ &\quad upper\_message\_error(good\_senders, ideal, actual) \ . \end{aligned}$$

For inexact communication, validity is the proposition that for a fixed receiver, the sample determined by a fault–masking vote is bounded both above and below by the sampled values received from good senders, modulo error values ε_{l} and ε_{u}. We express this in terms of a middle–value selection.

$$
\begin{aligned}
&inexact\_validity(eligible\_senders, good\_senders, ideal, actual) \stackrel{\text{df}}{=} \\
&\quad \exists s_1.\, s_1 \in good\_senders \wedge \\
&\qquad ideal(s_1) - \varepsilon_{\text{l}} \preceq middle\_value(make\_bag(eligible\_senders, actual)) \wedge \\
&\quad \exists s_2.\, s_2 \in good\_senders \wedge \\
&\qquad middle\_value(make\_bag(eligible\_senders, actual)) \preceq ideal(s_2) + \varepsilon_{\text{u}}\ .
\end{aligned}
$$

The Inexact Validity Theorem then reads:

Theorem 3 (Inexact Validity)
$majority\_good(good\_senders, eligible\_senders) \wedge$
$inexact\_message\_error(good\_senders, ideal, actual)$
implies that
$inexact\_validity(eligible\_senders, good\_senders, ideal, actual)$.

8 Conclusion

This paper presents, in the language of higher–order logic, four kinds of abstractions for fault–tolerant distributed systems. These abstractions pertain to messages, faults, fault–masking, and communication. We believe that they abstract a wide–variety of fault–tolerant distributed systems.

Other useful abstractions have been developed, too. For example, Rushby presents a set of conditions that guarantee that a time–triggered system implements a synchronous system [26]. These conditions have been used in the specification and verification of the Time–Triggered Architecture [22].

Our abstractions have proved their merit in an industrial–scale formal specification and verification project. We are sure that similar projects will profit. We are developing a distributed fault–tolerance library as part of the SPIDER project. It is designed to be a generic library of PVS theories that may be used in the specification and verification of a wide variety of fault–tolerant distributed systems. The abstractions described in this paper form the backbone of the library.

Abstractions support not only the design, specification, and verification, but also the documentation, comparison, and assessment of fault–tolerant distributed systems. We must learn to teach hardware and software engineers, system architects, certifiers, etc., the formalisms and the abstractions for these systems. We believe that abstractions can significantly shorten the time needed to acquire an understanding of the specifications as well as deepening that understanding. Being able to explain formal specification and verification to non–experts in formal methods is the first step to integrating formal proofs into the development life cycle.

Acknowledgments

We would like to thank Victor Carreño and Kristen Rozier for helpful comments.

References

1. Mohammad H. Azadmanesh and Roger M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, 2000.
2. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *ACM Proceedings: Operating Systems Design and Implementation (OSDI)*, pages 173–186, February 1999.
3. Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), February 1991.
4. Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 235–248. The 22nd International Conference on Computer Safety, Reliability and Security SAFECOMP, Springer-Verlag Heidelberg, September 2003.
5. Alfons Geser and Paul Miner. A formal correctness proof of the SPIDER diagnosis protocol. Technical Report 2002-211736, NASA Langley Research Center, Hampton, Virginia, August 2002. Technical Report contains the Track B proceedings from Theorem Proving in Higher Order Logics (TPHOLSs).
6. NASA LaRC Formal Methods Group. NASA Langley PVS libraries. Available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.
7. Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
8. SRI International. PVS homepage. Available at `http://pvs.csl.sri.com/`.
9. SRI International. PVS language reference, version 2.4. Available at `http://pvs.csl.sri.com/manuals.html`, December 2001.
10. Steven D. Johnson. Formal methods in embedded design. *Computer*, pages 104–106, Novemeber 2003.
11. Philip Koopman, editor. *Critical Embedded Automotive Networks*, volume 22-4 of *IEEE Micro*. IEEE Computer Society, July/August 2002.
12. Hermann Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.
13. Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982.
14. Leslie Lamport. Composition: A way to make proofs harder. *Lecture Notes in Computer Science*, 1536:402–423, 1998.
15. Jean-Claude Laprie. Dependability—its attributes, impairments and means. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictability Dependable Computing Systems*, ESPRIT Basic Research Series, pages 3–24. Springer, 1995.
16. Patrick Lincoln and John Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.
17. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
18. Thomas F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 129–157, Boston, 1988. Kluwer Academic Publishers.

19. Thomas F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.
20. Paul S. Miner, Mahyar Malekpour, and Wilfredo Torres-Pomales. Conceptual design of a Reliable Optical BUS (ROBUS). In *21st AIAA/IEEE Digital Avionics Systems Conference DASC*, Irvine, CA, October 2002.
21. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
22. Holger Pfeifer. *Formal Analysis of Fault-Tolerant Algorithms in the Time-Triggered Architecture*. PhD thesis, Universität Ulm, 2003. Available at `http://www.informatik.uni-ulm.de/ki/Papers/pfeifer-phd.html`.
23. Lee Pike. PVS specifications and proofs for fault-tolerant distributed system verification. Available at `http://shemesh.larc.nasa.gov/fm/spider/tphols2004/pvs.html`, 2004.
24. Maxwell Rosenlicht. *Introduction to Analysis*. Dover Publications, Inc., 1968.
25. John Rushby. Formal methods and digital systems validation for airborne systems. Technical Report CR–4551, NASA, December 1993.
26. John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September/October 1999.
27. John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001. Available at `http://www.csl.sri.com/~rushby/abstracts/buscompare`.
28. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Reliable Distributed Systems Symposium*, pages 93–100, October 1988.